# Table of contents

## Introduction

## System Design

## Application Architecture

## DOCA Libraries

## Compiling the Application

- Compiling All Applications
- Compiling Only the Current Application
- Troubleshooting

## Running the Application

- Prerequisites
- Application Execution
- Command Line Flags
- Troubleshooting

## Application Code Flow

## References
This guide provides process introspection system implementation on top of NVIDIA® BlueField® DPU.

**Introduction**

App Shield Agent monitors a process in the host system using the DOCA App Shield library.

This security capability helps identify corruption of core processes in the system from an independent and trusted DPU. This is a major and innovate intrusion detection system (IDS) ability since it cannot be provided from inside the host.

The **DOCA App Shield Library** gives the capability to read, analyze, and authenticate the host (bare metal/VM) memory directly from the DPU.

Using the library, this application hashes the un-writeable memory pages (also unloaded pages) of a specific process and its libraries. Then, at regular intervals, the app authenticates the loaded pages.

The app reports pass/fail after every iteration until the first attestation failure. The reports are both printed to the console and exported to the **DOCA Telemetry Service (DTS)** using inter-process communication (IPC).

This guide describes how to build secure process monitoring using the DOCA App Shield library, which leverages the DPU's advantages such as hardware-based DMA, integrity, and more.

**System Design**

The App Shield agent is designed to run independently on the DPU's Arm without hindering the host.

The host's involvement is limited to configuring monitoring of a new process when there is a need to generate the needed ZIP and JSON files to pass to the DPU. This is done at inception ("time 0") which is when the host is still in a "safe" state.

Generating the needed files can be done by running DOCA App Shield's `doca_apsh_config.py` tool on the host. See **DOCA App Shield** for more info.
Application Architecture

The user creates three mandatory files using the DOCA tool `doca_apsh_config.py` and copies them to the DPU. The application can report attestation results to the:

- File
- Terminal
- DTS

1. The files are generated by running `doca_apsh_config.py` on the host against the process at time zero.
2. The App Shield agent requests new attestation from DOCA App Shield library.

3. The DOCA App Shield library creates a new attestation:
   1. Scans and hashes process memory pages (that are currently in use).
   2. Compares the hash to the original hash.
   3. Creates attestation for each lib/exe involved in the process. Each of attestation includes the number of valid pages and the number of pages.

4. The App Shield agent searches each attestation for inconsistency between number of used pages and number of valid pages.

5. The App Shield agent reports results with a timestamp and scan count to:
   1. Local telemetry files – a folder and files representing the data a real DTS would have received. These files are used for the purposes of this example only as normally this data is not exported into user-readable files.
   2. DOCA log (without scan count).
   3. DTS IPC interface (even if no DTS is active).

6. The App Shield agent exits on first attestation failure.

**DOCA Libraries**

This application leverages the following DOCA libraries:

- [DOCA App Shield](#)
- [DOCA Telemetry Exporter](#)

Refer to their respective programming guide for more information.
Compiling the Application

The installation of DOCA's reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

**Tip**

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: /opt/mellanox/doca/applications/app_shield_agent/.

Compiling All Applications

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build
ninja -C /tmp/build
```
Compiling Only the Current Application

To build only the App Shield Agent application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_all_applications=false -Denable_app_shield_agent=true
ninja -C /tmp/build
```

Alternatively, the user can set the desired flags in the `meson_options.txt` file instead of providing them in the compilation command line:

1. Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
   - Set `enable_all_applications` to `false`
   - Set `enable_app_shield_agent` to `true`

2. Run the following compilation commands:
```
cd /opt/mellanox/doca/applications/
meson /tmp/build
```
Troubleshooting

Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the compilation of the application.

Running the Application

Prerequisites

1. Configure the BlueField's firmware.

   1. On the BlueField system, configure the PF base address register and NVMe emulation. Run:

   ```
dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 s PF_BAR2_SIZE=2 PF_BAR2_ENABLE=1
      NVME_EMULATION_ENABLE=1
   ```

   2. Perform a BlueField system reboot for the mlxconfig settings to take effect.

   3. You may verify these configurations using the following command:

   ```
dpu> mlxconfig -d /dev/mst/mt41686_pciconf0 q | grep -E "NVME|BAR"
   ```
2. Download target system (host/VM) symbols.

- For Ubuntu:

```bash
host> sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ $(lsb_release -cs) main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-updates main restricted universe multiverse
deb http://ddebs.ubuntu.com/ $(lsb_release -cs)-proposed main restricted universe multiverse
EOF
host> sudo apt install ubuntu-dbgsym-keyring
host> sudo apt-get update
host> sudo apt-get install linux-image-$(uname -r)-dbgsym
```

- For CentOS:

```bash
host> yum install --enablerepo=base-debuginfo kernel-devel-$(uname -r) kernel-debuginfo-$(uname -r) kernel-debuginfo-common-$(uname -m)-$(uname -r)
```

- No action is needed for Windows

3. Perform IOMMU passthrough. This stage is only necessary if IOMMU is not enabled by default (e.g., when the host is using an AMD CPU).

**Note**

Skip this step if you are not sure whether it is needed. Return to it only if DMA fails with a message similar to the following in `dmesg`:

```bash
host> dmesg
```
1. Locate your OS's grub file (most likely /boot/grub/grub.conf, /boot/grub2/grub.cfg, or /etc/default/grub) and open it for editing. Run:

```bash
host> vim /etc/default/grub
```

2. Search for the line defining `GRUB_CMDLINE_LINUX_DEFAULT` and add the argument `iommu=pt`. For example:

```bash
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt <intel/amd>_iommu=on"
```

3. Run:

```
host> sudo update-grub
host> ipmitool power cycle
```

**Note**

Prior to performing a power cycle, make sure to do a **graceful shutdown**.

- **For Ubuntu:**

  ```bash
  host> grub2-mkconfig -o /boot/grub2/grub.cfg
  ```

- **For CentOS:**

  ```bash
  host> grub2-mkconfig -o /boot/grub2/grub.cfg
  ```
4. Prepare target:

1. Install DOCA on the target system.

2. Create the ZIP and JSON files. Run:

   ```
   target-system> cd /opt/mellanox/doca/tools/
target-system> python3 doca_apsh_config.py --pid <pid-of-process-to-monitor> --os <windows/linux> --path <path to dwarf2json executable or pdbparse-to-json.py>
target-system> cp /opt/mellanox/doca/tools/*.* <shared-folder-with-baremetal>
dpu> scp <shared-folder-with-baremetal>/* <path-to-app-shield-binary>
   ```

   If the target system does not have DOCA installed, the script can be copied from the BlueField.

   The required `dwarf2json` and `pdbparse-to-json.py` are not provided with DOCA.

   ![Note]
   
   If the kernel and process `.exe` have not changed, there no need to redo this step.

### Application Execution

1. The App Shield Agent application is provided in source form; hence a compilation is required before the application can be executed.

   1. Application usage instructions:
Usage: doca_app_shield_agent [DOCA Flags] [Program Flags]

DOCA Flags:
- -h, --help                        Print a help synopsis
- -v, --version                     Print program version information
- -l, --log-level                   Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
- --sdk-log-level                   Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
- -j, --json <path>                 Parse all command flags from an input json file

Program Flags:
- -p, --pid                         Process ID of process to be attested
- -e, --ehm <path>                  Exec hash map path
- -m, --memr <path>                 System memory regions map
- -f, --vuid                        VUID of the System device
- -d, --dma                         DMA device name
- -o, --osym <path>                 System OS symbol map path
- -s, --osty <windows|linux>        System OS type - windows/linux
- -t, --time <seconds>              Scan time interval in seconds

Info
This usage printout can be printed to the command line using the -h (or --help) options:

```
./doca_app_shield_agent -h
```

Info
For additional information, please refer to section "Command Line Flags".
2. CLI example for running the application on the BlueField:

```
./doca_app_shield_agent -p 13577 -e hash.zip -m mem_regions.json -o symbols.json -f MT2125X03335MLNXS0D0F0VF1 -d mlx5_0 -t 3 -s linux
```

**Note**

All used identifiers (-f, -p and -d flags) should match the identifier of the desired devices and processes.

---

**Command Line Flags**

<table>
<thead>
<tr>
<th>Flag Type</th>
<th>Short Flag</th>
<th>Long Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General flags</td>
<td>h</td>
<td>help</td>
<td>Print a help synopsis</td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>version</td>
<td>Print program version information</td>
</tr>
<tr>
<td></td>
<td>l</td>
<td>log-level</td>
<td>Set the log level for the application:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- DISABLE=10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- CRITICAL=20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ERROR=30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- WARNING=40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- INFO=50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- DEBUG=60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- TRACE=70 (requires compilation with TRACE log level support)</td>
</tr>
<tr>
<td>N/A</td>
<td>sdk-log-level</td>
<td>Set the log level for the program:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DISABLE=10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- CRITICAL=20</td>
<td></td>
</tr>
<tr>
<td>Flag Type</td>
<td>Short Flag</td>
<td>Long Flag</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- ERROR=30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- WARNING=40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- INFO=50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- DEBUG=60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- TRACE=70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program flags</th>
<th>PID</th>
<th>Path to the pre-generated hash.zip file transferred from the host</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>pid</td>
<td>PID of the process to be attested</td>
</tr>
<tr>
<td>e</td>
<td>ehm</td>
<td>Path to the pre-generated hash.zip file transferred from the host</td>
</tr>
<tr>
<td>m</td>
<td>memr</td>
<td>Path to the pre-generated mem_regions.json file transferred from the host</td>
</tr>
<tr>
<td>f</td>
<td>pcif</td>
<td>System PCIe function vendor unique identifier (VUID) of the VF/PF exposed to the target system. Used for DMA operations. To obtain this argument, run:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-system&gt; lspci -vv</td>
</tr>
</tbody>
</table>

Example output:

```
[VU] Vendor specific: MT2125X03335MLNX50D0F0
[VU] Vendor specific: MT2125X03335MLNX50D0F1
```

Two VUIDs are printed for each DPU connected to the target system. The first is of the DPU on pf0 and the second is of the DPU on port pf1.

**Note**

Running this command on the DPU outputs VUIDs with an additional "EC" string in the middle. You must remove the "EC" to arrive at the correct VUID.
<table>
<thead>
<tr>
<th><strong>Flag Type</strong></th>
<th><strong>Short Flag</strong></th>
<th><strong>Long Flag</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>The VUID of a VF allocated on PF0/1 is the VUID of the PF with an additional suffix, ( \text{VF}&lt;\text{vf-number}&gt; ), where ( \text{vf-number} ) is the VF index +1. For example, for the output in the example above:</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>dma</td>
<td>• PF0 VUID = MT2125X03335MLNX50D0F0</td>
</tr>
<tr>
<td></td>
<td>o</td>
<td>osy</td>
<td>• PF1 VUID = MT2125X03335MLNX50D0F1</td>
</tr>
<tr>
<td></td>
<td>s</td>
<td>osty</td>
<td>• VUID of VF0 on PF0 = MT2125X03335MLNX50D0F0VF1</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>time</td>
<td>VUIDs are persistent even on reset.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d</th>
<th>dma</th>
<th>DMA device name to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>osy</td>
<td>Path to the pre-generated <code>symbols.json</code> file transferred from the host</td>
</tr>
<tr>
<td>s</td>
<td>osty</td>
<td>OS type (<code>windows</code> or <code>linux</code>) of the system where the process is running</td>
</tr>
<tr>
<td>t</td>
<td>time</td>
<td>Number of seconds to sleep between scans</td>
</tr>
</tbody>
</table>

**Info**

Refer to [DOCA Arg Parser](#) for more information regarding the supported flags and execution modes.

**Troubleshooting**

Refer to the [NVIDIA DOCA Troubleshooting Guide](#) for any issue encountered with the installation or execution of the DOCA applications.

**Application Code Flow**

1. Parse application argument.
1. Initialize arg parser resources and register DOCA general parameters.

   doca_argp_init();

2. Register application parameters.

   register_apsh_params();

3. Parse the arguments.

   doca_argp_start();

2. Initialize DOCA App Shield lib context.

   1. Create lib context.

      doca_apsh_create();

   2. Set DMA device for lib.

      doca_devinfo_list_create();
      doca_dev_open();
      doca_devinfo_list_destroy();
      doca_apsh_dma_dev_set();

   3. Start the context

      doca_apsh_start();
apsh_system_init();
3. Initialize DOCA App Shield lib system context handler.

   1. Get the representor of the remote PCIe function exposed to the system.

      
      ```
      doca_devinfo_remote_list_create();
      doca_dev_remote_open();
      doca_devinfo_remote_list_destroy();
      ```

   2. Create and start the system context handler.

      
      ```
      doca_apsh_system_create();
      doca_apsh_sys_os_symbol_map_set();
      doca_apsh_sys_mem_region_set();
      doca_apsh_sys_dev_set();
      doca_apsh_sys_os_type_set();
      doca_apsh_system_start();
      ```

4. Find target process by `pid`.

   ```
   doca_apsh_processes_get();
   ```

5. Telemetry initialization.

   ```
   telemetry_start();
   ```

   1. Initialize a new telemetry schema.

   2. Register attestation type event.

   3. Set up output to file (in addition to default IPC).

   4. Start the telemetry schema.

   5. Initialize and start a new DTS source with the `gethostname()` name as source ID.
6. Get initial attestation of the process.

   doca_apsh_attestation_get();

7. Loop until attestation validation fail.

   doca_apsh_attst_refresh();
   /* validation logic */
   doca_telemetry_exporter_source_report();
   DOCA_LOG_INFO();
   sleep();

8. DOCA App Shield Agent destroy.

   doca_apsh_attestation_free();
   doca_apsh_processes_free();
   doca_apsh_system_destroy();
   doca_apsh_destroy();
   doca_dev_close();
   doca_dev_remote_close();


   telemetry_destroy();

10. Arg parser destroy.

    doca_argp_destroy();

References
- `/opt/mellanox/doca/applications/app_shield_agent/`